

Deep Learning

Introduction to [PyTorch](#) Part 1

Lukas Galke & Md Shamim Ahmed – Slides from Lucas Dysse

Fall 2024

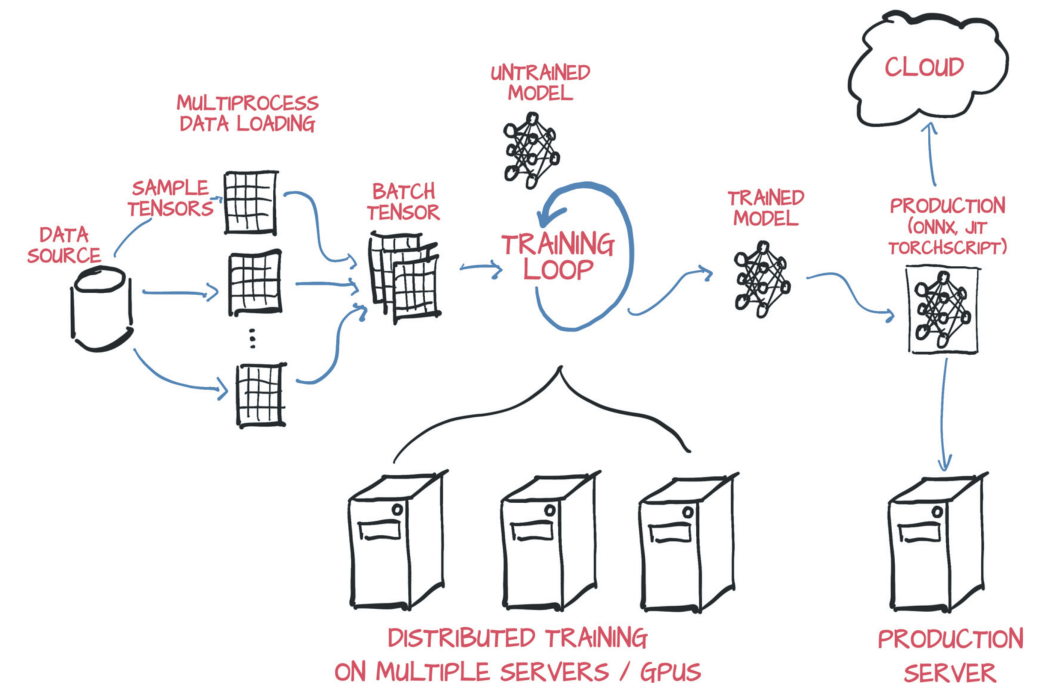
Introduction to PyTorch - Part I

- **What is PyTorch**
- **Building a Network**
- **Load data**
- **Train the model**
- **Evaluate the model**



What is PyTorch

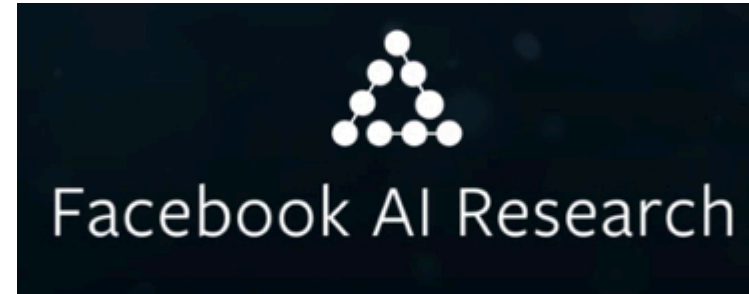
- PyTorch is an open-source deep learning framework.
 - It is a flexible and efficient tool for building and training neural networks.
- PyTorch offers seamless integration with GPUs, allowing for significant speed-ups in model training.
 - Run on CPU, NVIDIA GPU, (AMD GPU), Google TPUs...
- ONNX Support in PyTorch
 - PyTorch fully supports ONNX (Open Neural Network Exchange).
 - Export PyTorch models to ONNX format
 - Import pre-trained models in ONNX format into PyTorch
 - Enables conversion of PyTorch models to ONNX format for use in other frameworks, such as TensorFlow.




*PyTorch Deep Learning Framework: Speed + Usability | Synced. (2019, December 16).
Synced | AI Technology & Industry Review.
<https://syncedreview.com/2019/12/16/pytorch-deep-learning-framework-speed-usability/>*

Who makes PyTorch?

- Developed by Facebook's AI Research (FAIR) Lab
- Open-Source Community Collaboration
- 2022 Pytorch moved to Linux Foundation under the name PyTorch Foundation
- Examples of software built on top of PyTorch:
 - Tesla's Autopilot
 - Uber's Pyro
 - ...



The PyTorch user experience

- Intuitive and Pythonic API
 - PyTorch offers an intuitive and easy-to-use Pythonic API, making it user-friendly for researchers, developers, and data scientists.
- Dynamic Computation Graph
 - PyTorch's dynamic computation graph allows for immediate feedback and easy debugging, providing a more intuitive and flexible user experience compared to static graph frameworks.
 - We will discuss these points later, how PyTorch builds the computational graph on the fly 
- Flexible and explicit

Two API Styles

➤ **Imperative API** (Dynamic Computation Graph):

- The Imperative API is the default and most commonly used API in PyTorch.
- It allows you to define and execute operations on tensors dynamically as the code is being executed.
- With this dynamic computation graph, you can use standard Python control structures (e.g., loops and conditionals) to create complex models without explicitly defining the entire computational graph beforehand.
- The Imperative API is well-suited for research, prototyping, and experimentation due to its ease of use and flexibility.

➤ **Static API** (TorchScript and Tracing):

- The Static API in PyTorch, known as TorchScript, allows you to define and compile a model ahead of time.
- TorchScript uses a subset of Python syntax and enables you to create a static computational graph, similar to how it's done in static graph frameworks like TensorFlow.
- This static nature is useful for deployment scenarios where the model needs to be compiled once and then used for inference with optimized performance.
- The Static API provides more control and optimization opportunities for production deployments.

Introduction to PyTorch - Part I

- What is PyTorch
- **Building a Network**
- Load data
- Train the model
- Evaluate the model



PyTorch Neural Network

```
import torch
import torch.nn as nn

class MyNetwork(nn.Module):
    def __init__(self):
        super(MyNetwork, self).__init__()
        self.fc1 = nn.Linear(in_features=10, out_features=20)
        self.fc2 = nn.Linear(in_features=20, out_features=20)
        self.fc3 = nn.Linear(in_features=20, out_features=20)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.softmax(self.fc3(x))
        return x

net = MyNetwork()
```



- **nn.Module** is the base class for all neural network modules
- **__init__** initialize the weights, bias, etc.
 - **super(MyNetwork, self).__init__()** initialize the parent class (**nn.Module**)
 - Here, you define what components you gonna need
- **forward** passes the data through the network
 - forward is a default built-in function. You therefore do not need to write **net.forward(x)**, but can **net(x)**
 - The forward function of your network-class defines you the layers are connected to each other (you are assembling the computational graph)

Options for Layers

➤ Core Layers

- Convolutional Layers
- Pooling Layers
- Padding Layers
- Recurrent Layers
- Dropout Layers
- Normalization Layers
- Embedding Layers
- Transformer Layers
- Sparse Layers
- Vision Layers
- Shuffle Layers
- Data Parallel Layers

Options for Layers

- The core layers perform the most basic operations
- They are enough to built FFN Networks
- **Core Layers**
 - Linear Layers
 - Activation Layers
- We will learn to use many different layers
- Most layers can be easily dropped into the network

Linear Layer

- Linear layers in neural networks perform a linear transformation on the input data by applying weights and biases to produce output features.

```
torch.nn.Linear(in_features,      # Size of each input sample
                 out_features,     # Size of each output sample
                 bias=True,        # If set to False, the layer will not learn an additive bias. Default: True
                 device=None,     # An optional argument specifying the device (e.g., 'cpu' or 'cuda') where the tensor should be
                                # stored
                 dtype=None       # An optional argument specifying the data type of the weights and biases tensor
                 )
```

Activation Function

- Activation functions in neural networks introduce non-linearity to the model by applying a specific mathematical function element-wise to the output of a layer, allowing the network to model complex relationships and learn from non-linear patterns in the data.
- Pytorch include many activation functions including
 - softmax
 - elu
 - selu
 - softplus
 - relu
 - sigmoid
 - Tanh
- **inplace = True** means that we modify the input tensor directly without creating a new tensor for the output



```
relu = torch.nn.ReLU(inplace=False) # Can optionally do the operation in-place. Default: False
relu(input)
```

Introduction to PyTorch - Part I

- What is PyTorch
- Building a Network
- Load data
- Train the model
- Evaluate the model



Data Loader

```
DataLoader(dataset,
            batch_size=1,
            shuffle=False,
            sampler=None,
            batch_sampler=None,
            num_workers=0,
            collate_fn=None,
            pin_memory=False,
            drop_last=False,
            timeout=0,
            worker_init_fn=None,
            prefetch_factor=2,
            persistent_workers=False
            )
```

The dataset from which to load data (e.g., a TensorDataset or custom dataset).
The number of samples per batch to load. Default is 1 (i.e., no batching).
If True, shuffles the data at every epoch before creating batches.
Defines the strategy for sampling data indices, mutually exclusive with shuffle.
A custom batch sampler to specify how batches are created, mutually exclusive with batch_size, shuffle, and sampler.
The number of subprocesses to use for data loading. Default is 0 (loading data in the main process).
A function used to collate (merge) individual samples into batches (e.g., to pad variable-sized data). Default is None (using default collation).
If True, copies tensors into pinned memory, which can speed up data transfer to GPUs.
If True, drops the last incomplete batch if the dataset size is not divisible by the batch size.
The timeout value for the data loading processes, useful when using multiple workers.
A function that is called on each worker process at the beginning of data loading (e.g., setting random seeds).
Number of additional batches to prefetch (load in advance) for each worker.
If True, workers are kept alive after the loading process to reuse them in the next iteration. This can speed up data loading if data loading is the bottleneck.

Data Loader

```
from torch.utils.data import Dataset, DataLoader

batch_size = 32

train_dataset = CustomDataset(train_data, train_labels)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

#sdu.dk

➤ train_dataset needs to be a [CustomDataset](#) class. In this case we do not add any transformations.

```
class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

Load and Save models

➤ Saving a model

- You give it the parameters of a model using **MyModel.state_dict()** and a path of where you want to save the model including extension **.pth**

```
torch.save(MyModel.state_dict(), PATH)
```

➤ Loading a model

- You first need to initialize the network class
- Then you need to load a pretrained network by giving it the path including extension
- If you then want to use the model without training, you need to set it to evaluation mode by running **MyModel.eval()**

```
MyModel = MyNetwork()  
MyModel.load_state_dict(torch.load(PATH))  
MyModel.eval()
```

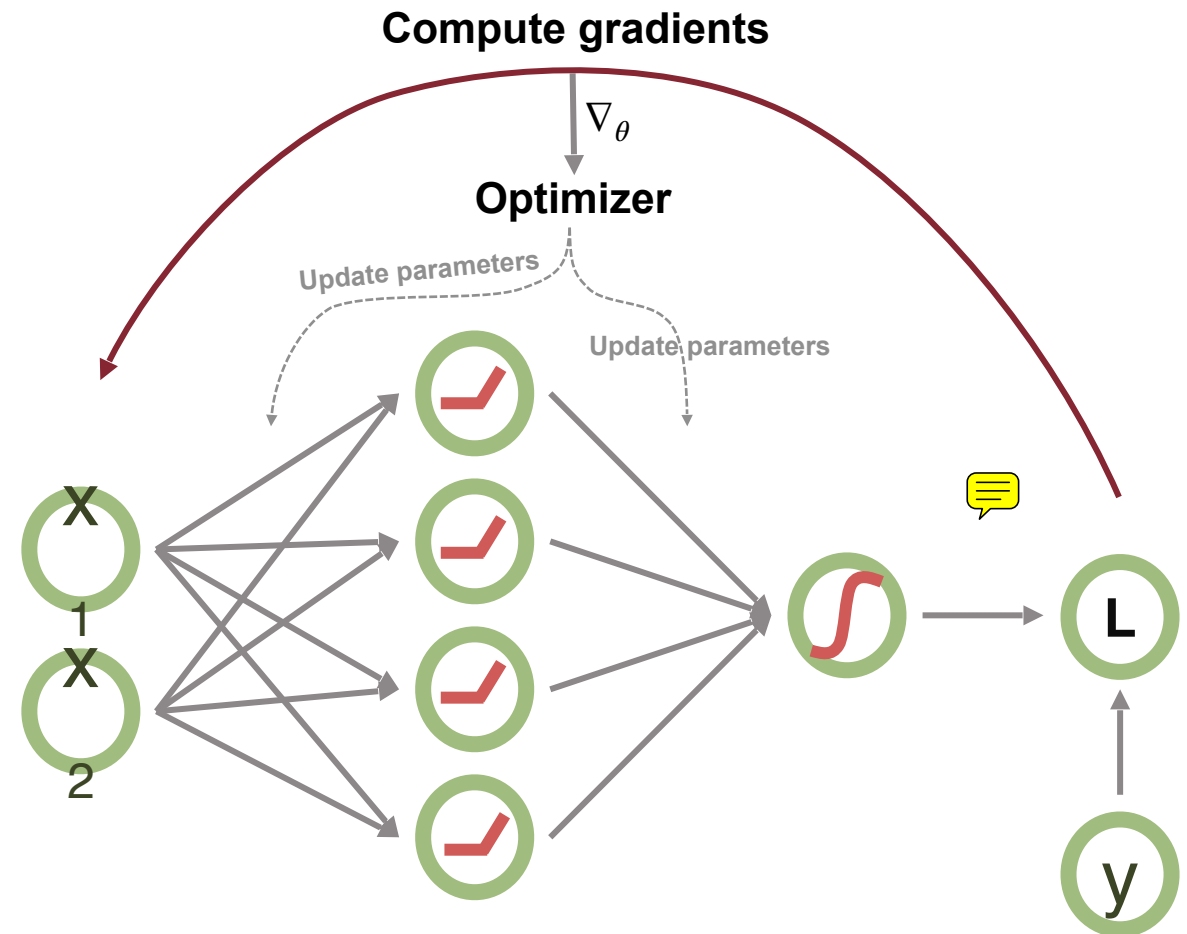

Introduction to PyTorch - Part I

- What is PyTorch
- Building a Network
- Load data
- **Train the model**
- Evaluate the model



The procedure of training a network

1. Define your network
2. Load your data
3. Create a minibatch
4. Perform Forward run
5. Compute loss
6. Compute gradient
7. Use your optimizer + gradient to update
8. Repeat 3 until abort criterion is matched



Loss Functions

- Loss functions in neural networks measure the discrepancy between the predicted output and the true target labels, guiding the model during training to minimize the error and improve its performance on the given task.
- L1Loss
- MSELoss
- CrossEntropyLoss
- BCELoss
- ...

```
criterion = torch.nn.MSELoss()  
loss = criterion(pred, label)
```

Metrics

- Metrics are evaluation criteria used to assess the performance of the model after training, quantifying how well the model performs on a specific task
- Can be any of the loss functions
- Some standard metrics like
 - F1
 - Precision
 - Recall
 - Accuracy

Optimizer

- Optimizers are algorithms used to adjust the model's parameters during training, aiming to minimize the loss function and find the optimal set of weights that best fit the data
- The most commonly used optimizers are:
 - SGD
 - ADAM

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

- Different optimizers take different inputs
 - The most common are learning rate (**lr**) and momentum
- Optimizers also take the model parameters (weights and bias) as input

Train the Model

```
model = MyNetwork()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(),
lr=0.01)

for epoch in range(num_epochs):
    model.train()
    for data, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
```

- When training the model, you want train multiple times on the data
 - Each training run through the data is called 1 epoch
- In each epoch you want to go through every piece of data in the dataloader
 - Remember, data in this case is a batch of data (eq. 64 images of cats and dogs). Labels are the class for each of those 64 images
- Remember you need to set **model.train()** when training for each epoch
 - This is due to some layers behaving different during training

Train the Model

```
model = MyNetwork()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(),
lr=0.01)

for epoch in range(num_epochs):
    model.train()
    for data, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
```

- **optimizer.zero_grad()** resets the gradients of all the model parameters
- **model(data)** passes a batch of data through the network and comes with predictions on each data piece
- **criterion** calculates using the defined loss function the loss between the outputs and the ground truth
- **loss.backward()** performs backpropagation to calculate the gradients with respect to the model's parameters
- **optimizer.step()** updates the parameters with respect to the gradients

Introduction to PyTorch - Part I

- What is PyTorch
- Building a Network
- Load data
- Train the model
- **Evaluate the model**



Evaluating the Model

```
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, targets in test_loader:
        outputs = model(data)
        _, predicted = torch.max(outputs.detach(),
dim=1)
        total += targets.size(0)
        correct += (predicted ==
targets).sum().item()

accuracy = 100 * correct / total
```

- When evaluating the performance of your model you need to set **model.eval()**
 - This changes the mode of the model to evaluation mode
- with **torch.no_grad()** disables gradient calculation, preventing PyTorch from tracking operations and using memory for backpropagation.
 - We only need the gradients when training, when evaluating we are not going to update any parameters
- **torch.max()** computes the maximum value and the corresponding index (class label) along an axis
- **outputs.detach()** detaches the tensor from the computation graph, making it independent of any future operations and gradient calculations

Evaluating the Model

```
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, targets in test_loader:
        outputs = model(data)
        _, predicted = torch.max(outputs.detach(),
dim=1)
        total += targets.size(0)
        correct += (predicted ==
targets).sum().item()

accuracy = 100 * correct / total
```

- **total += targets.size(0)** increments by the number of samples
- **correct += (predicted == targets).sum().item()** does the same except only for the cases where the network predicted correctly
- **item()** converts the results to a Python integer rather than a tensor

Plotting the training and validation loss

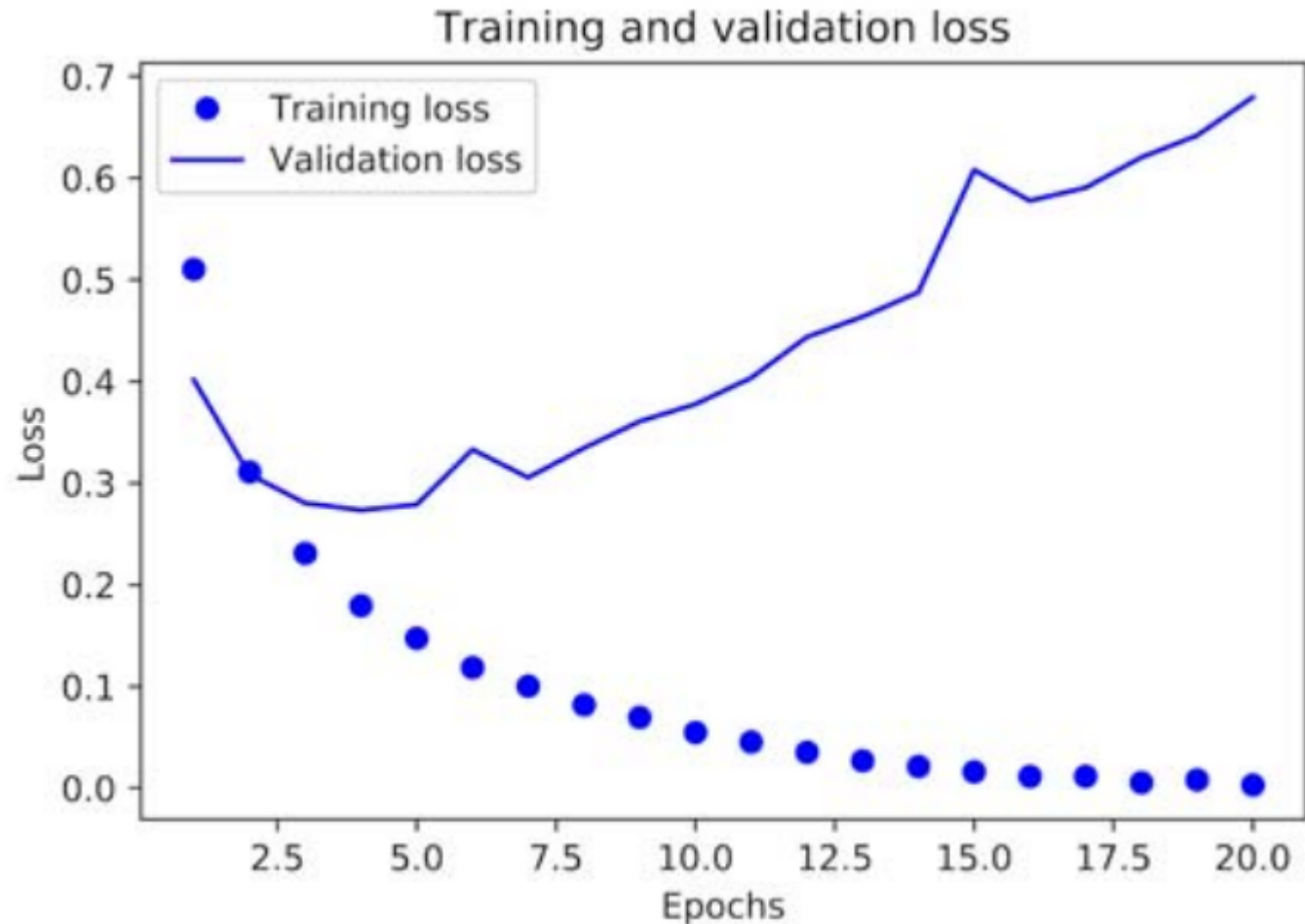
```
import matplotlib.pyplot as plt

plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation
loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

- `plt.plot` creates a line plot where data points are connected by a straight line.
 - It takes x-coordinates, y-coordinates, and different appearance arguments such as label, color, marker style, linestyle, etc.
 - In this case 'bo' is for blue dot, 'b' is for solid blue line
- In this case we are plotting both the training loss and the validation loss, we give it a title, x-label, y-label, and legends
- **`plt.show()`** shows the images
- If you want to save the image you can use **`plt.savefig()`** which takes a path ('/path/to/directory/plot.png')
 - Do before **`plt.show()`**

Plotting the training and validation loss



Practical Recommendations

- For lower data amounts, you should train smaller and shallower networks in order to prevent overfitting
- Preprocessing
 - Take small values - Typically, most values should be in the 0–1 range.
 - Be homogenous - That is, all features should take values in roughly the same range.